STA 32 R Handout 3, Functions in R

1 More Useful Functions

Note: the following assumes you have

```
X = c(67,63,65,71,72,68,60,59,67,66,75,74,69)
Y = c("F","F","F","M","M","M","F","F","M","F","M","M","F")
DataSet = data.frame(X,Y)
```

in your R session.

- `names`: We may want to name columns of a data frame (for example, when the .txt file had no header). To do this, we may use the command:

  ```
  names(DataSet) = c("Height","Gender")
  ```

  We must set the names of the data set equal to a character vector which has the same number of elements as the data set does columns. Since `DataSet` had two columns, I had to assign the names of `DataSet` to a character vector of length two. Now we may use the commands `DataSet$Gender`, for example.

- `which`: We have already seen the which function, but now we may be interested in combining two logical commands. Say we want to view all females who are above 65 inches in height. That is, we want to list people who are female, and whose height is above 65 inches. We may combine two logical checks with the & command:

  ```
  DataSet[DataSet$Gender == "F" & DataSet$Height >65,]
  ```

  We could also create an "or" command, using the vertical bar —:

  ```
  DataSet[DataSet$Height <62 | DataSet$Height >70,]
  ```

  Now we have listed all the data points where the height is either greater than 70, or less than 62. You can put any of this commands into a `which` statement to give back the row indices that match the logical arguments.

  ```
  which(DataSet$Gender == "F" & DataSet$Height >65)
  which(DataSet$Height <62 | DataSet$Height >70)
  ```

- `order`: The function order can be used on a vector to return the row index such that the numbers would be sorted in increasing order (by default). If I wanted to reorder the data set so that the height was increasing, I would use the following command:

  ```
  order(DataSet$Height) #Gives the order the rows should be in
  DataSet = DataSet[order(DataSet$Height),] #Reorders the data set
  ```

  Notice if I leave the columns blank, then the entire rows are reordered, not a single column in the dataset. To change the order to decreasing, add the argument `decreasing = TRUE` to the `order` function.

- `round`: Takes in a vector (or a number), and the number of digits you want to round to. For example,

  ```
  TheMean = mean(DataSet$Height)
  round(TheMean,digits = 0)
  ```

  rounds the mean to the nearest whole number.

- `ceiling`: Takes in a vector or number, and rounds all numbers **up**.

  ```
  ceiling(TheMean)
  ```

- `floor`: Takes in a vector or number, and rounds all numbers **down**.

  ```
  floor(TheMean)
  ```

- `par(mfrow = c(1,1))`: This function sets the number of rows and columns to fill in with plots on one page. It is defaulted to one row, one column, i.e. one plot per page. If we wanted two plots per page, we would give `R` the command

  ```
  par(mfrow = c(2,1))
  hist(DataSet$Height,main = "Histogram of Height", xlab = "Height in Inches")
  boxplot(DataSet$Height,main = "Boxplot of Height", ylab = "Height in Inches")
  ```

  which would say we want two plots on the same page, in column format (i.e two rows, one column). The `par(mfrow = ... )` command must be used BEFORE you use any plotting command.

- `sample`: Takes in a vector of characters or numbers, the number of times you want to sample from that vector, if you want to sample with replacement or not (defaults to without replacement), and the probabilities associated with each element of the vector (defaults to equal probability). For example, to sample 1:10 five times without replacement, and to sample from the character strings "Cat","Bird", "Dog" ten times with replacement where the probabilities of sampling are (.45,.05,.50), respectively:

```
FirstEx = sample(1:10,5)
SecondEx=sample( c("Cat","Bird","Dog"), 10, replace = TRUE, prob = c(.45,.05,.50))
```

- `rep`: takes in a number, character vector, or numeric vector, the number of times you want the whole vector repeated, or the number of times you want each element in the vector repeated. For example, repeating the value 0 twenty times, repeating the vector `c(''All'', ''the'', ''things!'')` four times, repeating the values 1:4 each 4 times, and repeating 2 twice, 4 four times, and 6 six times (in that order) can be done by:

```
Ex1=rep(0,20)
Ex2=rep(c("All","the","things!"),times = 4)
Ex3=rep(1:4, each = 4)
Ex4=rep( c(2,4,6), times = c(2,4,6))
```

- `set.seed`: set the seed for random number generator. It's a habit to always set a seed so others can reproduce your result:

```
set.seed(1010)
```

2 Making Your Own Functions

Frequently, there may not be a function in `R` that does exactly what you want. Fortunately we are able to create our own functions, as follows:

```
SampleFun = function(input1, input2, ...){
   #Body of the function, i.e. commands to run
   return( #your result)
}
```

Functions should have arguments that are passed to them (in the same way you pass a vector to the mean function) and should return the result specified by the code inside the function. You would then call your function using the name, for example by `SampleFun(input1,input2)`
To make a function to find the range of the data, one could do the following:

```
MyRange = function(X){
   n = length(X) #Setting the number of points total
   OrderedX = X[order(X)] #Reordering the X vector
   #Finding the min and max
   Min = OrderedX[1]
   Max = OrderedX[n]
   Range = Max - Min  #Calculating the range
   return(Range) #Outputting the desired result
}
MyRange(DataSet$Height)
```

A critical requirement of functions is that any variable used in the function should either be passed in as an input argument, or be defined in the function itself For example, I passed in the vector `X`, and then in the function itself I defined `n` to be the length of `X`. If I had not defined `n` inside of the function `MyRange`, it would have given back an error (hopefully!).
A function can only return **one** object, however, that object could be a vector of numbers. For example, if I wanted `MyRange` to give back the min, max, and range (in that order), I could change the return statement to: `return(c(Min,Max,Range))`

2.1 The `if` Statement

An `if` statement checks to see if the condition you give it is true, and if so, runs the code you put in the body of the function. For example,

```
Do = "Add"
X = 2
Y = 4
if(Do == "Add"){
   Z = X+Y
}
```

Only if the variable `Do` is equal to the character string "`Add`" will the variable `Z` be created. We can use an `if` statement into a function as well:

```
Toyfun = function(X,Y,Do){
   if(Do == "Add"){
      Z = X+Y
      return(Z)
   }
   return(c(X,Y))
}
Toyfun(2,4,"Add")
Toyfun(2,4,"NoAdd")
```

Since in the second call to `ToyFun`, the `Do` argument was not "`Add`", the function never entered the body of the if statement where it would create `Z`, and instead simply returned `X` and `Y` like it was specified to.
We can also use the `else` command after the body of an `if` statement to go through multiple checks:

```
Toyfun = function(X,Y,Do){
   if(Do == "Add"){
      Z = X+Y
      return(Z)
   } else if(Do =="Subtract"){
      Z = X-Y
      return(Z)
```

```
   }else if(Do =="Multiply"){
      Z = X*Y
      return(Z)
   }else if(Do =="Penguin"){
      return(c("<(’’ )"))
   }
      return(c(X,Y))
}

Toyfun(2,4,"Add")
Toyfun(2,4,"Subtract")
Toyfun(2,4,"Penguin")
```

3 `sapply` and `for` loops

When you want to run the same piece of code multiple times (in a simulation, for example), we have two options. We may use the `sapply` function, where we give it the number of times we want to run the code (from 1 to some number `i`), the index (which may change values of a parameter for each run), and the function we want to run. This is, in general, faster than using a `for` loop, where you specify for i in 1 to some number, do the following commands (and typically, you save the result as the ith element of a vector.
An example where we wish to flip a fair coin 100 times, and return a `TRUE` if the flip was heads, and a `FALSE` if it was tails using both a `for` loop and `sapply` follows:

```
#Using the sapply function:
Results = sapply(1:100,function(i){
   Flip = sample(c("H","T"),1,replace = TRUE)
   if(Flip =="H"){return(TRUE)} else{return(FALSE)}
})
#Using a for loop:
Results1 = rep(NA,100) #Initializing the result vector
for(i in 1:length(Results1)){
   Flip = sample(c("H","T"),1,replace = TRUE)
   if(Flip =="H"){
      Results1[i] = TRUE
      } else{
      Results1[i] = FALSE
      }
}
```

For the `sapply` statement: We first define the number of times we want to run this function, from 1 to 100. Then, we define a function (which takes in i) that is to be run for each iteration. Notice this function (at each iteration) will return either a `TRUE` or a `FALSE`, and there is no need to initialize a vector and fill in the results of the experiment.
For the `for` loop, the first thing you must do is initialize the vector, and the for loop will go

through and update the elements of the vector with the result of the function fun. In the body of the for loop, for each iteration we flip a coin, and if it is heads the i^th element of the `Results1` vector is set equal to `TRUE`, otherwise it is set equal to `FALSE`.
In `R`, `sapply` statements are in general faster than `for` loops, and I encourage you to only use `sapply` statements in your homework. If I wanted to make a function that

took in the value n and outputted the probability of a head or tail for n runs, I could do the following:

```
HeadsOrTails = function(n){
   Results = sapply(1:n,function(i){
      Flip = sample(c("H","T"),1,replace = TRUE)
      if(Flip =="H"){return(TRUE)} else{return(FALSE)}
   })
   ProbOfHeads = sum(Results)/length(Results)
   return(ProbOfHeads)
}
HeadsOrTails(10)
HeadsOrTails(100)
HeadsOrTails(1000)
HeadsOrTails(10000)
HeadsOrTails(100000)
```

Note: the larger n is, the more simulated examples we took, and the closer the probability should be to the true probability of 0.50.

4 More Examples

4.1 Example 1:

A function which identifies outliers:

```
#Finds outliers, with an option to remove them from the dataset.
FindOutliers = function(Data,column = 1,Remove = FALSE){
  if(!is.numeric(Data[,column])) stop("Input is not numeric")
  X = Data[,column]
  Quant = fivenum(X)
  Q1 = Quant[2]
  Q3 = Quant[4]
  IQR = Q3-Q1
  LFence = Q1 - 1.5*IQR
  UFence = Q3 + 1.5*IQR
  Outlier = which(X < LFence | X > UFence)
  if(length(Outlier) != 0){
    print(paste("Observation", Outlier,"is an outlier."))
    if(Remove == TRUE){
```

```
      Data = Data[,-Outlier]
      print(paste("Observation",Outlier,"with value",X[Outlier],"is removed"))
      return(Data)
    }
    return(Outlier)
  }else{
    print("There are no outliers.")
  }
}
```

Key points: If you set default values for arguments inputted to your function, the function will use those values unless you specify them as something different.
You can nest if statements.
If `which` finds no observations that match the logical conditions, it returns `integer(0)`, which is not the same as the value `0`. `integer(0)` has length 0.
`paste` is used to put together character stings and numerics. `print` prints out in the console the value given it.
The `stop` function is serve as a check point. If the input is not a numeric vector, R will stop the function and return an error message.

4.2 Example 2:

A function which checks to see if two randomly drawn cards without replacement have the same suit, does this experiment n times, and returns the probability of drawing two cards and having the suit match.

```
#Draws two cards from a 52 card deck, and checks to see if they are the same suit.
SuitCheck = function(n){
  Results = sapply(1:n,function(i){
    DeckValues = rep(c("H","S","C","D"),each = 13)
    Draw = sample(DeckValues,2,replace = FALSE)
    return(Draw[1] == Draw[2])
  })
  Prob = sum(Results)/length(Results)
  return(Prob)
}
```

4.3 Example 3:

A function which draws a single card from a deck, and calculates probabilities of based on n single draws.

```
#Draws one card from a 52 card deck, calculates many probabilities.
#Let A be the event that there is an Ace draw.
#Let B be the event that a heart is draw.
```

```
#Calculates probabilities based on the above.
SuitCheck = function(n){
  Suit = rep(c("H","S","C","D"),each = 13)
  Value = rep(c("A",2:10,"J","Q","K"),times = 4)
  Deck = cbind(Suit,Value)
  Results = sapply(1:n,function(i){
    Draw = sample(1:52,1,replace = FALSE)
    return(Deck[Draw,])
  })
  Results = t(Results)
  P.A = sum(Results[,2]=="A")/nrow(Results)
  P.B = sum(Results[,1] == "H")/nrow(Results)
  P.AandB = sum(Results[,2] == "A" & Results[,1] == "H")/nrow(Results)
  P.AorB = sum(Results[,2] == "A" | Results[,1] == "H")/nrow(Results)
  AllResults = matrix(c(P.A,P.B,P.AandB,P.AorB),nrow = 1)
  colnames(AllResults) = c("P(A)","P(B)","P(A&B)","P(AUB)")
  return(AllResults)
}
```

The last three lines in the function formats the results nicely. We made the results into a matrix with one row with the function `matrix`, and then we gave that matrix column names with the function `colnames`.